Assignment 2 - Solving two 1D problems

In [1]:
```
import numpy as np
%matplotlib inline
from matplotlib import pyplot as plt
from matplotlib import cm
import seaborn as sns
plt.style.use('seaborn-notebook')
```

Part 1: Solving a wave problem with sparse matrices

In this part of the assignment, we want to compute the solution to the following (time-harmonic) wave problem:

$$\begin{aligned} \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + k^2 u = 0 &\qquad \text{in } (0,1),\\ u = 0 &\qquad \text{if } x = 0,\\ u = 1 &\qquad \text{if } x = 1,\end{aligned}$$

with wavenumber $k = 29\pi/2$.

In this part, we will approximately solving this problem using the method of finite differences. We do this by taking an evenly spaced values $x_0 = 0, x_1, x_2, \dots, x_N = 1$ and approximating the value of u for each value: we will call these approximations u_i. To compute these approximations, we use the approximation

$$\frac{\mathrm{d}^2 u_i}{\mathrm{d}x^2} \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2},$$

where $h = 1/N$.

With a bit of algebra, we see that the wave problem can be written as

$$(2 - h^2k^2)u_i - u_{i-1} - u_{i+1} = 0$$

if x_i is not 0 or 1, and

$$\begin{aligned} u_i = 0 &\qquad \text{if } x_i = 0,\\ u_i = 1 &\qquad \text{if } x_i = 1.\end{aligned}$$

This information can be used to re-write the problem as the matrix-vector problem $\mathrm{A}\mathbf{u} = \mathbf{f}$, where A is a known matrix, $\mathbf{f}$ is a known vector, and $\mathbf{u}$ is an unknown vector that we want to compute. The entries of $\mathbf{f}$ and $\mathbf{u}$ are given by

$$\begin{aligned} [\mathbf{u}]_i &= u_i,\\ [\mathbf{f}]_i &= \begin{cases} 1 & \text{if } i = N,\\ 0 & \text{otherwise.}\end{cases}\end{aligned}$$

The rows of A are given by

$$[\mathrm{A}]_{i,j} = \begin{cases} 1 & \text{if } i = j,\\ 0 & \text{otherwise,}\end{cases}$$

if $i = 0$ or $i = N$; and

$$[\mathrm{A}]_{i,j} = \begin{cases} 2 - h^2k^2 & \text{if } j = i,\\ -1 & \text{if } j = i+1,\\ -1 & \text{if } j = i-1.\\ 0 & \text{otherwise,}\end{cases}$$

otherwise.

a)

Write a Python function that takes N as an input and returns the matrix A and vector f. You should use an appropriate sparse storage format for the matrix A.

In [2]:
```
from scipy.sparse import coo_matrix
```

In [3]:
```
def discretise_wave(N):
    """Generate the matrix and rhs associated with the discrete time-harmonic wave equatio
n."""

    h = 1/N
    k = 29*np.pi/2

    #Number of elements is N+1 for diagonal + the off diagonal (2N-2) with the top and bott
om row removed
    nelements = N+1 + 2*N - 2

    #Set up coo styled data storage
    row_ind = np.zeros(nelements, dtype=int)
    col_ind = np.zeros(nelements, dtype=int)
    data = np.zeros(nelements, dtype=np.float64)

    #Create the vector (all zeros with a 1 at the end)
    f = np.concatenate((np.zeros(N, dtype=np.float64),[1]))

    #Set the 0,0 element to be 1
    row_ind[0] = col_ind[0] = 0
    data[0] =  1
    #Set the N,N element to be 1
    row_ind[-1] = col_ind[-1] = N
    data[-1] =  1

    #Loop through each row excluding the top and bottom
    count = 1
    for i in range(1,N):

      #For each row we have three data entries
      row_ind[count : count+3] = i

      #For the diagonals we have 2-h^2k^2
      col_ind[count] = i
      data[count] = 2-h**2*k**2

      #For the off diagonals we have -1
      col_ind[count + 1] = i + 1
      col_ind[count + 2] = i - 1
      data[count + 1 : count + 3] = -1

      count += 3


    return coo_matrix((data, (row_ind, col_ind)), shape=(N+1, N+1)).tocsr(), f
```

A quick example, if we use N = 2 we expect to get a matrix out which looks like:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 2-h^2k^2 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

And our vector should be:

$$\mathbf{f} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

In [4]:
```
A, f = discretise_wave(2)
print(A.toarray())
print(f)
```

```
[[   1.            0.            0.        ]
 [  -1.         -516.77108133   -1.        ]
 [   0.            0.            1.        ]]
[0. 0. 1.]
```

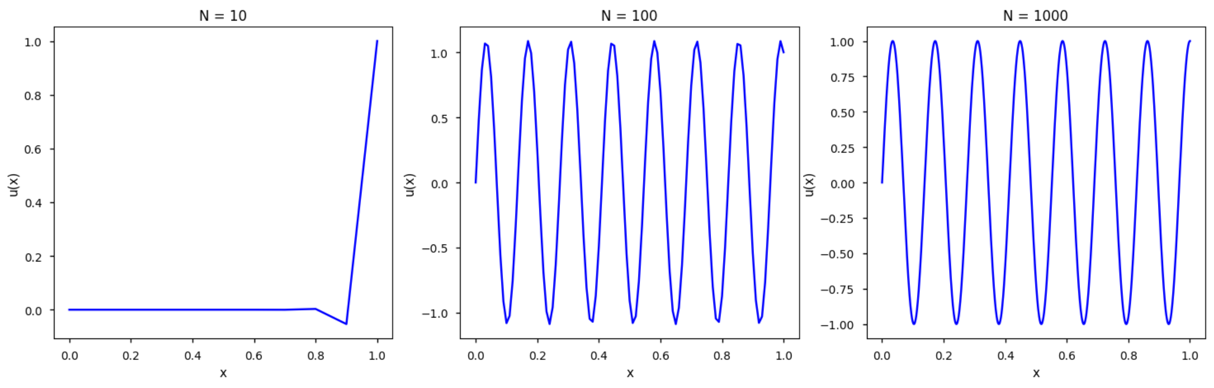
b)

The function `scipy.sparse.linalg.spsolve` can be used to solve a sparse matrix-vector problem. Use this to **compute the approximate solution for your problem for $N = 10, N = 100$, and $N = 1000$**. Use `matplotlib` (or any other plotting library) to **plot the solutions for these three values of N**.

In [5]:
```python
from scipy.sparse.linalg import spsolve
```

In [6]:
```python
sols = []

Ns = [10,100,1000]

for N in Ns:

  A, f = discretise_wave(N)
  sols.append(spsolve(A, f))
```

In [7]:
```python
fig, ax = plt.subplots(1,3,figsize=(18, 5),dpi=100)

for idx, N in enumerate(Ns):
  x = np.linspace(0,1,N+1)
  ax[idx].plot(x,sols[idx],c='b')
  ax[idx].set_ylabel("u(x)")
  ax[idx].set_xlabel("x")
  ax[idx].set_title(f"N = {N}")
```



c)

Briefly (1-2 sentences) comment on your plots: How different are they to each other? Which do you expect to be closest to the actual solution of the wave problem?

Answer - The N=10 solution is very different to the other two - it has not produced a wave. The other two solutions are closer to eachother, both show the same overall wave shape. The greater the value of N used, the smoother the function becomes and the closer I would expect it to be to the true analytical solution.

d)

This wave problem was carefully chosen so that its exact solution is known: this solution is $u_{\text{exact}}(x) = \sin(kx)$. (You can check this by differentiating this twice and substituting, but you do not need to do this part of this assignment.)

A possible approximate measure of the error in your solution can be found by computing

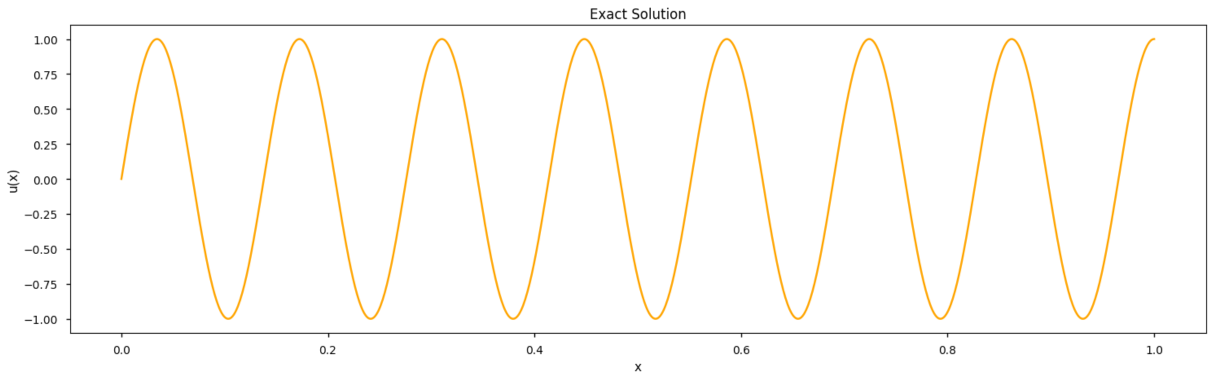
$$\max_i |u_i - u_{\text{exact}}(x_i)| .$$

Compute this error for a range of values for N of your choice, for the methods you wrote in both parts 1 and 2. On axes that both use log scales, **plot N against the error in your solution**. You should pick a range of values for N so that this plot will give you useful information about the methods.

In [8]:
```
def exact(x):

  k = 29*np.pi/2

  return np.sin(k*x)
```

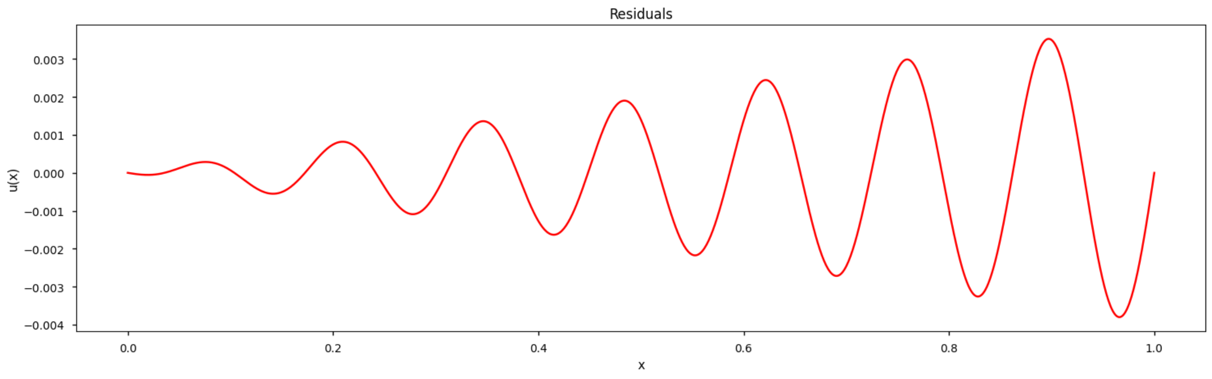
A plot to see the exact solution:

In [9]:
```
fig, ax = plt.subplots(1,figsize=(18, 5),dpi=100)
x = np.linspace(0,1,1001)
ax.plot(x,exact(x),c='orange')
ax.set_ylabel("u(x)")
ax.set_xlabel("x")
ax.set_title("Exact Solution");
```



A plot of the residuals between the solved values and exact values:

In [10]:
```
fig, ax = plt.subplots(1,figsize=(18, 5),dpi=100)
x = np.linspace(0,1,1001)
ax.plot(x,exact(x)-sols[2],c='red')
ax.set_ylabel("u(x)")
ax.set_xlabel("x")
ax.set_title("Residuals");
```



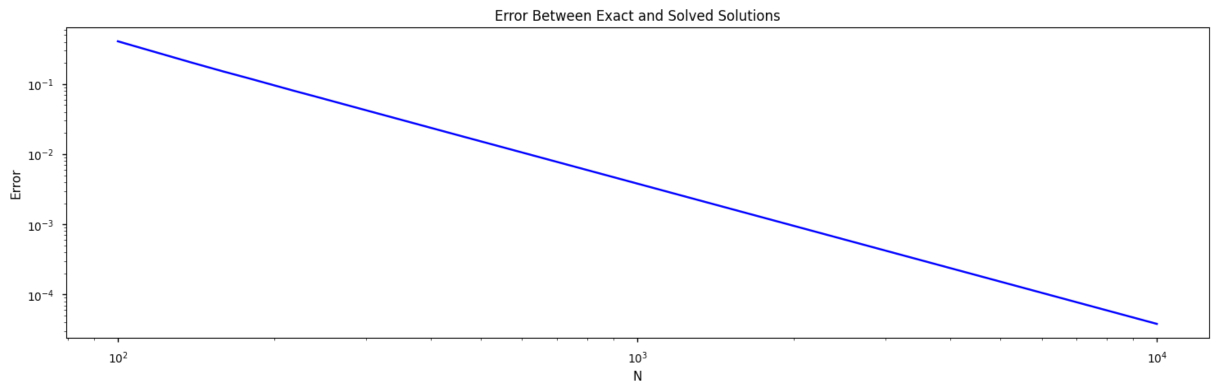
Next we define the error function:

In [11]:
```
def err(u,u_exact):

  return np.amax(np.absolute(u-u_exact))
```

And then check the error for increasing N:

```
In [42]: Ns = np.linspace(100,10000,500,dtype=int)
         errs = np.zeros(len(Ns))
         for idx, N in enumerate(Ns):

           #Solve the equations
           A, f = discretise_wave(N)
           sol = spsolve(A, f)

           #Calcualte the exact value
           ex = exact(np.linspace(0,1,N+1))

           #Calcualte the error
           errs[idx] = err(sol,ex)
```

```
In [43]: fig, ax = plt.subplots(1,figsize=(18, 5),dpi=100)
         ax.plot(Ns,errs,c='b')
         ax.set_ylabel("Error")
         ax.set_xlabel("N")
         ax.set_yscale("log")
         ax.set_xscale("log")
         ax.set_title("Error Between Exact and Solved Solutions");
```



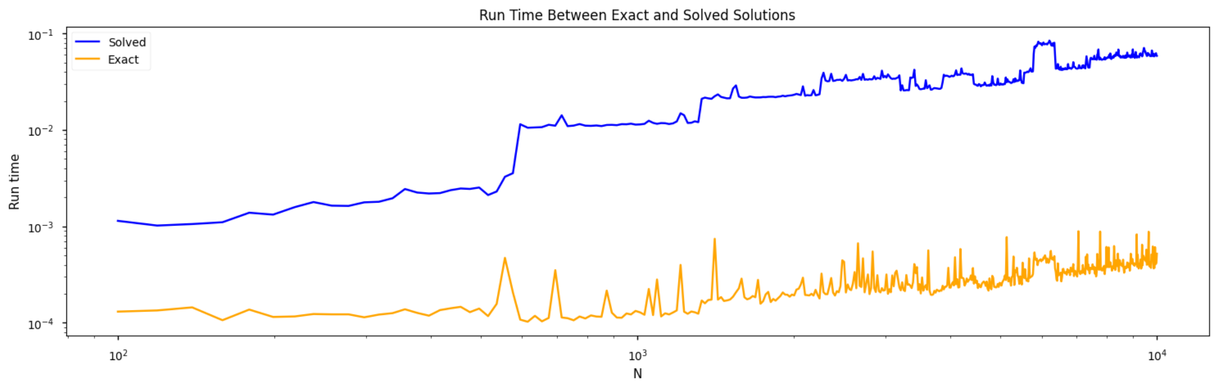
e)

For the same values of N, **measure the time taken to compute your approximations for both functions**. On axes that both use log scales, **plot N against the time taken to compute a solution**.

```
In [36]: from time import time
```

```
In [40]: Ns = np.linspace(100,10000,500,dtype=int)
         times = np.zeros((2,len(Ns)))
         for idx, N in enumerate(Ns):

           #Solve the equations
           start = time()
           A, f = discretise_wave(N)
           spsolve(A, f)
           times[0,idx] = time() - start

           #Calculate the exact value
           start = time()
           exact(np.linspace(0,1,N+1))
           times[1,idx] = time() - start
```

In [41]:
```python
fig, ax = plt.subplots(1,figsize=(18, 5),dpi=100)
ax.plot(Ns,times[0],label = "Solved",c='blue')
ax.plot(Ns,times[1] , label = "Exact",c='orange')
ax.set_ylabel("Run time")
ax.set_xlabel("N")
ax.set_yscale("log")
ax.set_xscale("log")
ax.set_title("Run Time Between Exact and Solved Solutions")
ax.legend();
```



f)

We now want to compute an approximate solution where the measure of error is 10^{-8} or less. By looking at your plots, **pick a value of N that you would expect to give error of 10^{-8} or less. Briefly (1-2 sentences) explain how you picked your value of N and predict how long the computation will take**.

Answer - I would choose a value of N of the order of maginutude $\approx 10^6$. This is because the order of maginutude of the error roughly decreases linearly with the order of magnitude of N such that:

$$\log(\text{Err}(N)) \approx -2\log(N) + c$$

At $N \approx 10^4$ we have an error of $\approx 10^{-4}$, so for $N \approx 10^6$ I would expect an error of $\approx 10^{-8}$. As for run time, it seems like this is linked to N roughly (with a few steps along the way) by:

$$\log(\text{Time}(N)) \approx \log(N) + d$$

Where c and d are constants. I would expect a run time around order of maginutude 10^0 seconds as at $N \approx 10^4$ we have a run time of around $\approx 10^{-1}$.

g)

Compute the approximate solution with your value of N. Measure the time taken and the error, and **briefly (1-2 sentences) comment on how these compare to your predictions**.

In [17]:
```python
def time_and_error(N):

    #Solve the equations
    start = time()
    A, f = discretise_wave(N)
    sol = spsolve(A, f)
    run_time = time() - start

    #Calculate the exact value
    ex = exact(np.linspace(0,1,N+1))

    #Calculate the error
    er = err(sol,ex)

    print(f"N: {N}")
    print(f"Error: {er}")
    print(f"Runtime: {run_time}")
```

```
In [18]: time_and_error(int(1e6))
```

```
N: 1000000
Error: 1.5517885523438912e-08
Runtime: 2.2923946380615234
```

Answer - The error isn't quite as low as I predicted - I expected this as my estimates were only made very crudely by checking how many orders of maginitude the runtime or the error changed when I changed the order of magnitude of N - however, the order of magintude for both the error and run time are as I expected. Interestingly, if N is increased very slightly from 10^6, which I would expect to bring down the error, the error instead increases while the run time continues with the trend of increasing:

```
In [19]: time_and_error(int(1.00001e6))
         time_and_error(int(1e7))
```

```
N: 1000010
Error: 2.0754513803400432e-07
Runtime: 2.1744508743286133
N: 10000000
Error: 0.00011509646200105403
Runtime: 23.856406211853027
```

Answer continued - this is possibly due to the fact that as N increases, the value of h^2 decreases and so we are taking away a quadratically smaller number from 2. This may introduce some machine precision errors - a double is accurate up to 16 decimal places, when we use $N = 10^6$ we are generating a value of h^2 with the largest digit at 12 decimal places.

Part 2: Solving the heat equation with GPU acceleration

In this part of the assignment, we want to solve the heat equation

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= \frac{1}{1000}\frac{\mathrm{d}^2u}{\mathrm{d}x^2} \qquad \text{for } x \in (0,1), \\ u(x,0) &= 0, \\ u(0,t) &= 10, \\ u(1,t) &= 10. \end{aligned}$$

This represents a rod that starts at 0 temperature which is heated to a temperature of 10 at both ends.

Again, we will approximately solve this by taking an evenly spaced values $x_0 = 0, x_1, x_2, \ldots, x_N = 1$. Additionally, we will take a set of evenly spaced times $t_0 = 0, t_1 = h, t_2 = 2h, t_3 = 3h, \ldots$, where $h = 1/N$. We will write $u_i^{(j)}$ for the approximate value of u at point x_i and time t_j (ie $u_i^{(j)} \approx u(x_i, t_j)$).

Approximating both derivatives (similar to what we did in part 1), and doing some algebra, we can rewrite the heat equation as

$$\begin{aligned} u_i^{(j+1)} &= u_i^{(j)} + \frac{u_{i-1}^{(j)} - 2u_i^{(j)} + u_{i+1}^{(j)}}{1000h}, \\ u_i^{(0)} &= 0, \\ u_0^{(j)} &= 10, \\ u_N^{(j)} &= 10. \end{aligned}$$

This leads us to an iterative method for solving this problem: first, at $t = 0$, we set

$$u_i^{(0)} = \begin{cases} 10 & \text{if } i = 0 \text{ or } i = N, \\ 0 & \text{otherwise;} \end{cases}$$

then for all later values of time, we set

$$u_i^{(j+1)} = \begin{cases} 10 & \text{if } x = 0 \text{ or } x = N, \\ u_i^{(j)} + \dfrac{u_{i-1}^{(j)} - 2u_i^{(j)} + u_{i+1}^{(j)}}{1000h} & \text{otherwise.} \end{cases}$$

a)

Implement this iterative scheme in Python. You should implement this as a function that takes N as an input.

```
In [44]: import numba
```

My function below takes the temperature values of a rod, $u(x)$ discretised into N+1 indexes, and updates them by iterating through the heat equation up to some maximum time, t_{max}. The total number of time steps is given by $t_{max} \times N$:

```
In [96]: @numba.njit(['int32,int32,float64[:,:]'])
         def solve_heat(N,tmax,u):

           #Loop through times
           for j in range((N*tmax)):
             #Update the points in the rod excluding the end points
             u[j+1,1:N] = u[j,1:N] + (u[j,0:N-1] - 2*u[j,1:N] + u[j,2:N+1])*N/1000
```

If memory was an issue (in the case of a large t_{max}) I would have the function above only update a 1D N+1 long array and not store the rod values for each timestep. As memory is not too much of an issue in this scenario, I save all of the time step data points as this makes plotting simple. Let's try this for $N = 3$ and $t_{max} = 2$, I would expect an array out which has $N + 1$ columns (with 10 at the ends) and then $N \times t_{max} + 1$ rows, one row for each time step plus the $t = 0$ row:

```
In [97]: N = 3
         tmax = 2

         #Create the rod and set the ends to a constant temperature
         u = np.zeros((N*tmax+1,N+1))
         u[:,0] = u[:,-1] = 10

         solve_heat(N,tmax,u)
         u

Out[97]: array([[10.        ,  0.        ,  0.        , 10.        ],
                [10.        ,  0.03      ,  0.03      , 10.        ],
                [10.        ,  0.05991   ,  0.05991   , 10.        ],
                [10.        ,  0.08973027,  0.08973027, 10.        ],
                [10.        ,  0.11946108,  0.11946108, 10.        ],
                [10.        ,  0.1491027 ,  0.1491027 , 10.        ],
                [10.        ,  0.17865539,  0.17865539, 10.        ]])
```

b)

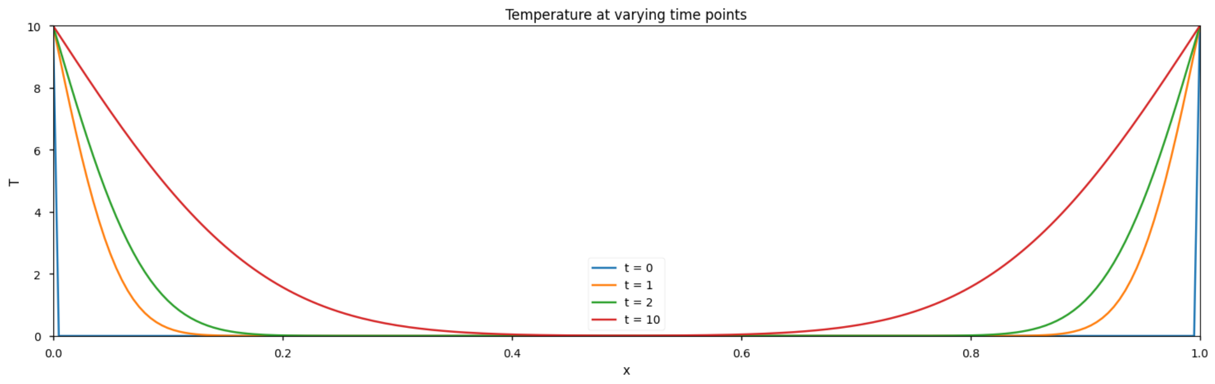
Using a sensible value of N, **plot the temperature of the rod at $t = 1$, $t = 2$ and $t = 10$. Briefly (1-2 sentences) comment on how you picked a value for N.**

Answer - I have used a value of N which provides me with enough x values to give a smooth plot while being low enough to run quickly. Also I have made sure N is below 500 - if N goes above 500 it seems to break the function.

```
In [87]: N = 200
         tmax = 10

         #Create the rod and set the ends to a constant temperature
         u = np.zeros((N*tmax+1,N+1))
         u[:,0] = 10
         u[:,-1] = 10

         solve_heat(N,tmax,u)
```

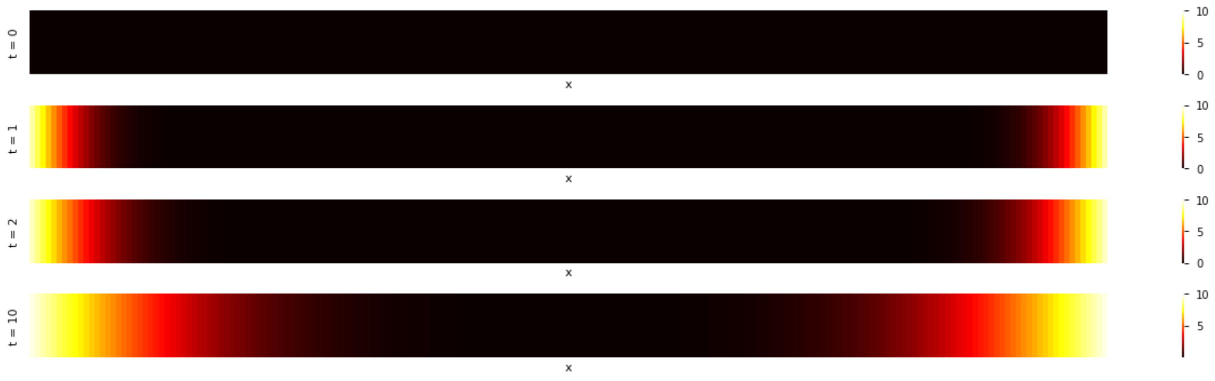
Below is a plot showing the temperature across the rod for different time values:

In [88]:
```
fig, ax = plt.subplots(1,figsize=(18, 5),dpi=100)

ts = [0,1,2,10]
for t in ts:
  ax.plot(np.linspace(0,1,N+1),u[int(t*N)],label=f't = {t}')

ax.set_ylabel("T")
ax.set_xlabel("x")
ax.set_xlim([0,1])
ax.set_ylim([0,10])
ax.set_title("Temperature at varying time points")
ax.legend();
```



A heatmap visulisation of the rod:

In [49]:
```
fig, ax = plt.subplots(4,1,figsize=(18, 5))
for idx,t in enumerate(ts):
  sns.heatmap(u[int(t*N)][np.newaxis,:], cmap='hot',ax=ax[idx])
  ax[idx].set_xlabel("x")
  ax[idx].set_ylabel(f"t = {t}")
  ax[idx].set_yticks([])
  ax[idx].set_xticks([])
fig.tight_layout()
```



c)

Use `numba.cuda` to parallelise your implementation on a GPU. You should think carefully about when data needs to be copied, and be careful not to copy data to/from the GPU when not needed.

In [98]:
```python
from numba import cuda
cuda.detect()
```

```
Found 1 CUDA devices
id 0             b'Tesla T4'                              [SUPPORTED]
                      Compute Capability: 7.5
                           PCI Device ID: 4
                              PCI Bus ID: 0
                                    UUID: GPU-6ed858f0-bdbd-0850-2d14-6dc30f598ba4
                                Watchdog: Disabled
             FP32/FP64 Performance Ratio: 32
Summary:
        1/1 devices are supported
```

Out[98]:
```
True
```

We want to parallelise the process of updating each point on the bar:

In [99]:
```python
#Our kernel function - iterates through the bar
@cuda.jit
def solve_heat_cuda(u0,u1):

  # Index of thread on GPU
  i = cuda.grid(1)

  #Get N from the size of the array
  N = u0.size-1

  #Update the points in the rod excluding the end points
  if(i > 0 and i < N):
    #Update the vector for the next time point
    u1[i] = u0[i] + (u0[i + 1] - 2*u0[i] + u0[i - 1])*N/1000
```

In [115]:
```python
N = 3
tmax = 2

#Create the rod and set the ends to a constant temperature
u = np.zeros((N+1))
u[0] = u[-1] = 10

# Manage the number of threads
threadsperblock = 32
blockspergrid = (u.size + (threadsperblock - 1)) // threadsperblock

#Copy the rod to the GPU plus a buffer to be updated
u0 = cuda.to_device(u)
u1 = cuda.to_device(u)

#Update the rod
for j in range(N*tmax):
  if (j % 2) == 0:
    solve_heat_cuda[blockspergrid, threadsperblock](u0,u1)

  else:
    solve_heat_cuda[blockspergrid, threadsperblock](u1,u0)

#Copy the final timestep data back to us
u = u0.copy_to_host()
print(u)
```

```
[10.          0.17865539  0.17865539 10.        ]
```

GPU vs CPU Runtime:

In [104]:
```
N = 300
tmax = 200

##CPU BASED##
uCPU = np.zeros((N*tmax+1,N+1))
uCPU[:,0] = uCPU[:,-1] = 10

start = time()
solve_heat(N,tmax,uCPU)
print(f"CPU Runtime: {time()-start}")
######################################

##GPU BASED##
uGPU = np.zeros((N+1))
uGPU[0] = uGPU[-1] = 10

# Manage the number of threads
threadsperblock = 32
blockspergrid = (uGPU.size + (threadsperblock - 1)) // threadsperblock

#Add the rod to the GPU
u0 = cuda.to_device(uGPU)
u1 = cuda.to_device(uGPU)

start = time()
#Update the rod
for j in range(N*tmax):
  if (j % 2) == 0:
    solve_heat_cuda[blockspergrid, threadsperblock](u0,u1)

  else:
    solve_heat_cuda[blockspergrid, threadsperblock](u1,u0)

#Ensure the GPU code is finished before we get the end time
cuda.synchronize()

print(f"GPU Runtime: {time()-start}")
print(f"Total number of threads: {threadsperblock*blockspergrid}")
#Copy the final timestep data back to us
uGPU = u0.copy_to_host()

############################################
print(f"Function difference: {np.sum(uGPU-uCPU[-1])}")
np.testing.assert_allclose(uGPU,uCPU[-1])
```

```
CPU Runtime: 0.04960227012634277
GPU Runtime: 3.367361545562744
Total number of threads: 320
Function difference: 0.0
```

d)

Use your code to estimate the time at which the temperature of the midpoint of the rod first exceeds a temperature of 9.8. Briefly (2-3 sentences) describe how you estimated this time. You may choose to use a plot or diagram to aid your description, but it is not essential to include a plot.

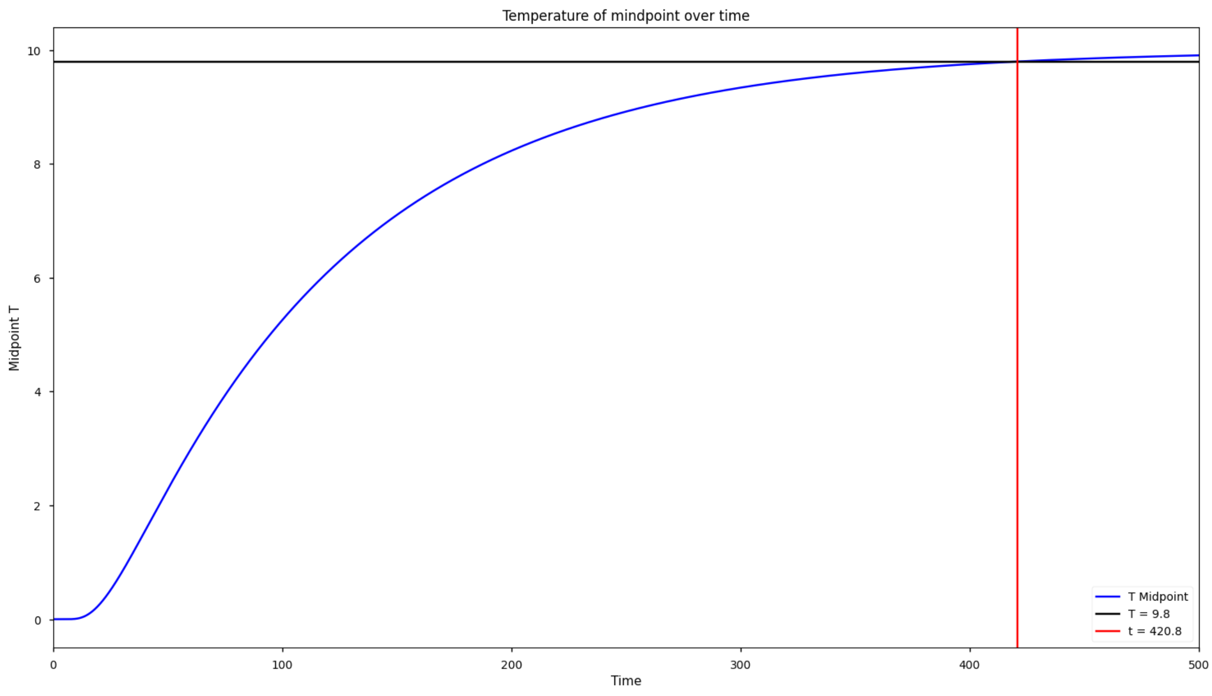
Answer - For this question I used my CPU based code which returns to me all values of the rod up to a maximum time t_{max}. I then check to see at what time step the mid point, index $\mathrm{int}(\frac{N+1}{2})$, first exceeded 9.8 and return that time step converted into seconds:

In [116]:
```
def find_time(u,T):

  N = u.shape[1] - 1

  #Find all the mid points
  mid_points = u[:,int((N+1)/2)]

  try:
    #Get the first mid point value which goes above T - convert from timestep into seconds
    timeT = np.asarray(mid_points > T).nonzero()[0][0]/N
    print(f"Mid point temperature > {T} found at t = {timeT}")
    return timeT

  except:
    print(f"Mid point temperature > {T} not found")
```

In [125]:
```
#Run the heat equation for a large tmax
N = 200
tmax = 500

#Create the rod and set the ends to a constant temperature
u = np.zeros((N*tmax+1,N+1))
u[:,0] = 10
u[:,-1] = 10

#Solve the heat equation
solve_heat(N,tmax,u)

#Check to see if we've found a midpoint temperature of 9.8
timeT = find_time(u,9.8)
```

```
Mid point temperature > 9.8 found at t = 420.845
```

In [126]:
```
fig, ax = plt.subplots(1,figsize=(18, 10),dpi=100)
ax.plot(np.linspace(0,tmax,N*tmax+1),u[:,int((N+1)/2)],label = 'T Midpoint',c='b')
ax.set_ylabel("Midpoint T")
ax.set_xlabel("Time")
ax.set_title("Temperature of mindpoint over time")
ax.set_xlim([0,tmax])
ax.axhline(9.8,c='black',label = 'T = 9.8')
ax.axvline(timeT,c='red',label = f't = {timeT:0.1f}')
ax.legend();
```



Answer Continued - If I was to do this using my GPU based code, I would run the time stepping loop until a midpoint temperature of 9.8 was identified. I could also use my array of CPU generated mid point values and then create an array with an equal number of points going from 0 to t_{max}, interpolation could then be used to identify the time at which the temperature is equal to 9.8.

In [127]:
```
print(f"Interpolated time at which midpoint = 9.8: {np.interp(9.8, u[:,int((N+1)/2)], np.li
nspace(0,tmax,N*tmax+1))}")
```

```
Interpolated time at which midpoint = 9.8: 420.8425859423344
```